Architecture of Large-Scale Systems

Arne Koschel, Irina Astrova, Elena Deutschkämer, Jacob Ester, Johannes Feldmann

***Abstract*—**In this paper various techniques in relation to large-scale systems are presented. At first, explanation of large-scale systems and differences from traditional systems are given. Next, possible specifications and requirements on hardware and software are listed. Finally, examples of large-scale systems are presented.

***Keywords*—**Distributed file systems, cashing, large scale systems, MapReduce algorithm, NoSQL databases.

I. Introduction

TODAY there are not only software systems running on a single client, but there are also systems running on servers on which multiple clients can access. For the latter there are different requirements, including scalability.

There are two types of scalability: vertical and horizontal. Vertical scalability means that the original system will be replaced with a more powerful system. Horizontal scalability means that the original system remains but extra servers are added. For large-scale systems the vertical scalability is of particular interest [4].

A. What are Large-Scale Systems?

There is no single exact definition of a large-scale system. However, there are criteria to define such a system:

- The amount of data which is processed. "Processed"
- means here storing, accessing, manipulating, and refining
- The number of hardware elements
- The number of people who are involved
- The number of systems, which are purposed and processed.

For both traditional and large-scale systems, the main points are performance, reliability, complexity, development and process. But these points have to be scalable in large-scale systems, so that the whole system works as a unit.

B. Traditional Systems vs. Large-Scale Systems

A good analogy to describe the differences between traditional and large-scale systems is the comparison of differences among a house, a high-rise building and a city.

A large-scale system is usually 'a system of systems'; it can be viewed as a city with many houses and high-rise buildings. That means a large-scale system has a high number of functions. The functions are expanding over time like a city is growing over time. In a traditional system the number of functions is constant. There are updates and small extensions, but there are not large extensions of the functions. This is comparable with a house or a high-rise building. A house is a more persistent object and its lifetime changes are usually limited to small improvements or additions.

As with a city, the architecture of the system is not clearly defined at the beginning. There are always changes because of different enhancements. For a house, the architecture is planned in the beginning, also for a traditional system. For a city, the architecture is also planned in the beginning, but because of different conditions at later times the architecture may need to be changed. The same applies to large-scale systems. The architecture cannot be defined in the beginning because it does not endure over the whole lifecycle of the system. That means a large-scale system has to be flexible for changes because of expansions. A traditional system can be more static. Not only because of the growing number of functions, but also because of the rising user count, a large-scale system has to be flexible related to the scalability. This time the scalability and flexibility aspects of a traditional system are more considered as a projection of users to computers. For a large-scale system there always has to be the possibility to scale the system for more users [1].

Table I presents the most important characteristics of traditional and large-scale systems.

TABLE I
CHARACTERISTICS OF TRADITIONAL AND LARGE-SCALE SYSTEMS [1]

Characteristic	Traditional system	Large-scale system
Governance	Singular dominant influence	Multiple, conflicting influences
Duration of life	Defined at the moment of designing	Infinite
Flow of information	Well-understood internal flow, known sources	Changing flow of information, new sources
Size	Local	Often global
Boundaries	probably determined	Unknown, changeable, fluctuating
Complexity	Optimized	Highly complex, not optimized
Elements	Services, components	Systems, services
Constructor	Own organization or COTS	COTS or foreign

II. Requirements for Large-Sale Systems

In general, defining hardware and software for large-scale systems is not very trivial. This fact is mainly due to the large number of possible applications and scenarios for large-scale systems. The requirements for hardware and software are largely dependent on the particular application. The hardware requirements for a social network like Facebook compared to a large-scale-scientific-cluster for calculating weather data are different. To create/develop a solid requirements specification from the beginning is difficult due to a prospective and continuous evolution of the system itself. Regardless of these

Irina Astrova is with the Institute of Cybernetics, Tallinn University of Technology, Estonia (e-mail: irina@cs.ioc.ee).

Arne Koschel, Elena Deutschkämer, Jacob Ester, and Johannes Feldmann are with the Faculty IV, Department for Computer Science, University of Applied Sciences and Arts Hannover, Hannover Germany (e-mail: arne.koschel@hs-hannover.de).

difficulties there are hardware and software, which enforced on the marked for the usage in large-scale systems. Compared with traditional systems the error handling is very important. In a traditional system the failure of any single component often affects the whole system. On the contrary, failures of components in a large-scale system are considered and planned in the lifecycle of the system. Hardware and software should be designed to handle the failure of any single component. Facebook, but also Intel and HP, have spent a lot of time developing specialized hardware and software for large-scale systems. Recently Facebook presented their research in finding effective hardware in the "Open Computer Project" (http://opencompute.org/). Within this research, a group of Facebook employees has written instructions and specifications, which gave hints for the construction of very efficient servers.

However, there are various software approaches, which are specially designed for large-scale systems. Especially for web-based large-sale systems there are lots of implementations. Best known of them are probably the Map-Reduce algorithm, more precisely Apache Hadoop, which is an implementation based on the later described map-reduce algorithm. Other often sees tools are specializations of Memcached, a cache server, which reduces the load on nodes in the large-scale system. Through the use of such software, the performance of large-scale systems can be generally increased significantly, because the load on the single individual node is reduced as the content is available in the cache, or the flood of requests is efficiently distributed across multiple servers, which act as clusters. There are also implementations that ensure the availability of the system even when some parts of the system fail.

A very good example, as mentioned earlier, for a large-scale system is the Internet itself. The architecture is decentralized and the development is carried out continuously and evolutionary. If some parts of the Internet, e.g. a single server or a route fails, the rest of the Internet acts without errors as nothing happened, because there are alternative routes and pages (servers) that can be accessed by the users.

III. Components of Large-Scale Systems

In the following, there will be given explanations of the basic components for large-scale systems.

A. File Systems

In traditional systems, file systems are used to organize files on a single hard disk drive or a RAID system. They provide applications access to files and directories by directly reading from and writing to a physical media.

Major software systems are distributed over several machines, which include the file systems used. Distributed file systems spread the system's data over multiple machines and disks. Many clients share the same data using one or more servers as point of access. An underlying network connects all machines which access the file system. They communicate using a certain protocol. Additional features like mechanisms for data replication or fault tolerance may be included on file system level. Prominent examples of conventional distributed file systems are Suns NFS, Apples AFP and SMB, which are mostly used within Microsoft Windows-driven environments. However, large-scale systems need to go one step further for several reasons such as performance, scalability and integration into existing infrastructure.

Google File System: This is an example of a distributed file system. When Google designed Google File System (GFS) [19] in 2003, the developers made a set of assumptions:

- For reasons of economy, Google chose to use commodity hardware, which is expected to fail. Therefore, a monitoring mechanism is needed, to ensure prompt recovery.
- GFS should be optimized for handling a huge number of large files, because they are used in experience mostly.
- In practical use, read operations are more common than write operations. At this, reads are sequential or random, while writes are supposed to be sequential rather than random. There are practically no over-write operations.
- Files act as producer-consumer queues with extensive merging. This means prevalently appending to one certain file. That's why atomicity is essential while producing minimal overhead in synchronization.
- As observed in other applications, reading a huge amount of bulk data is more common than tasks, which depend on low latency. Therefore a high sustained bandwidth is more relevant than low response times.

These assumptions are based on Google's very own, former experience in developing large-scale systems and high performance solutions. This shows GFS was highly optimized for large-scale system needs.

As shown in Fig. 1, a GFS cluster consists of one master and several chunk servers. Chunk servers store files in fixed-sized chunks, which are identified by a globally unique ID, the chunk handle. Chunks are stored on a commodity hard disk as Linux files. By default three replications of each chunk are stored on different chunk servers. The master manages all chunks and takes care about replication and metadata. Metadata includes access control information, mapping from files to chunk servers and the location of chunks. Master and chunk servers communicate periodically. Here, the state of chunk servers is collected and instructions are given by the master.

A cluster is accessed by multiple clients. When an application needs a file, the client API sends a request to the master. The master then provides metadata which enables the client to access the certain chunk server that holds the requested file. There is no payload data transferred between a client and the master. For reasons of performance, the master holds the entire metadata in its RAM. Further, a client can ask the server for multiple chunks metadata within one single request. Therefore in this architecture a single centralized master is not a bottleneck. As shown, the flow of control data and payload data is separated. This leads to an efficient access behavior, while still benefiting from one single master. There

is no overload produced by synchronizing several masters.

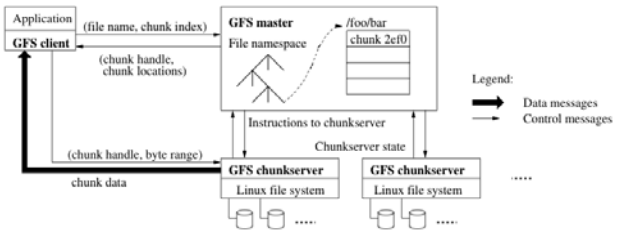

Fig. 1 Architecture of Google File System

As mentioned in the assumptions above, a large chunk size is desired. This is for several reasons: working with big chunks leads to less metadata to be stored on the master. Further, there is less communication between the client and the master to transfer the metadata. Since most read and write operations are sequential, the TCP/IP connection handling benefits from large chunks as connections can stay open over a long period of time. That's why GFS uses a chunk size of 64 MB, which is much larger than in traditional distributed file systems.

To provide atomic writes with a good performance, GFS provides an operation called *record append*. Using this operation, the client does not have to deal with the exact position of where to append new data to a chunk. Normally, this position is described by an offset that is sent to the client on request. When it comes to concurrent writes to one chunk by multiple clients, data may be overwritten. Clients may use offsets, which point to addresses that are already used by other client's data, since there is no concurrency management on the client-side. Therefore, in GFS the clients transfer only the appending data to the server.

The server manages concurrent writes and appends one client's data sequentially to the chunk at a position chosen by the server. Then the server sends back the actual position of the appended data within the chunk to the client for later access.

To keep a cluster available when hardware fails, which is common, GFS uses two mechanisms. These are fast recovery and replication. Fast recovery ensures minimal start-up times of master and chunk servers. If a process fails, a server will simply retry after a short time-out. Replication applies to both chunks and masters. Chunks are replicated three times by default. Replicas are managed by the master, including chunk verification using checksums. Master replication provides reliability of the cluster. Backup masters can easily recover a cluster's state by reading the log of a crash master.

Further, a master may be "shadowed" which can be considered as a delayed live-copy of a master. These shadows independently communicate with the chunk servers. In case of a crashed master, shadows can provide read-only access to applications that do not depend on up-to-data data. In addition, GFS offers features like garbage collection, directory snapshots and load balancing.

GFS is a highly optimized and customized solution. Its architecture has been designed to serve Google's specific demands on high performance and high availability while managing a huge amount of data. The architecture enables horizontal scalability by simply adding chunk servers to the cluster, which is one essential requirement on the technologies when applied in large-scale systems.

Amazon Simple Storage Service: This is another major distributed file system that is used in a large-scale system. Although as described above, the GFS is proprietary as well, Amazon does not provide detailed information on the internal architecture of Amazon Simple Storage Service (Amazon S3) (http://aws.amazon.com/s3/#functionality).

Amazon S3 is an online storage accessible through web services. It stores arbitrary objects that are organized in so called buckets. Objects may be up to 5 terabytes large and can be accessed with simple read and write operations. Buckets are available via several Internet protocols, among them HTTP and BitTorrent. REST and SOAP interfaces are available as well. Amazon S3 provides authentication mechanism to impede unauthorized access. The data buckets are located in specific geographical regions, which can be of interest when storing sensitive business data. Today, Amazon S3 stores about 450 billion objects and processes up to 290,000 requests a second at peak time [21].

Google developed a customized solution to run their in-house applications and systems. In contrast, Amazon uses its existing infrastructure and free resources to provide scalable memory to third parties using cloud services.

B. Databases

Like for a traditional system, for a large-scale system any database system can be used. But a new kind of databases has arisen: NoSQL databases [5][6]. The main difference to traditional (SQL) databases is that the focus is on scalability. Large-scale applications grow within time. There must be the possibility to scale up both the systems and the databases. NoSQL databases are normally characterized through weak schema restriction, so that it is easy to upgrade the data records. It also should be much easier to replicate the data in NoSQL databases than in traditional databases.

In most cases, in NoSQL databases ACID (Atomicity, Consistency, Isolation, Durability) is not the right concurrency control method. In this method a data record is locked until one operation has finished. This could be obstructive in large-scale systems, because such systems must be available at any time. At least they must receive read and write operations at any time. This means that availability and scalability are much more important than consistency. Instead of ACID, NoSQL databases often use BASE (Basically Available, Soft-State, Eventual Consistency). The idea is that consistency should be reached eventually, but availability is of the highest priority. Data records are not locked when they are in use.

There are three main types of NoSQL databases: column-oriented databases, document stores and key-value stores.

Column-oriented databases: These databases are called column-oriented because of the way how the data records are

persisted. Traditional databases are called line-oriented because all attributes are consecutive in one line of the table. In a column-oriented database every column can be persisted in a separate file. This means a value of a special attribute does not follow the value of the next special attribute of the same tuple. Instead, a value of a special attribute follows the value of the same attribute of the next tuple [7], [8].

The main disadvantage of traditional databases: If a sum over more than one line of the table should be formed, it needs computing time. This is not a problem for traditional systems as they are optimized for those operations. But it is a problem for a very large amount of data. In such systems a lot of computing time will need. Because of that, it makes sense to persist the sum every time when it changed. This happens essentially in column-oriented databases. Data records are stored in a way that values can be summed up with as little input/output activity as possible.

Google's Big Table (http://www.neogrid.de/was-ist/Google-BigTable) is an example of column-oriented databases. Google has to deal with a high volume of data, which lays in the dimension of petabytes. Because of that, Google developed BigTable. BigTable can be used for large amounts of structured data or for systems which need low response times. Several concepts are united in this database: scalability, high performance, low downtime.

The data model is a weak, distributed, persistent multidimensional sorted map. The index consists of a row key, a column key and a timestamp. Every value in the map is an uninterpreted array of bytes [3].

Row keys are arbitrary strings. Every writing or reading with a certain row-key is atomic, no matter how many rows or lines are addressed. So it is easier for the clients to justify the system behavior if there are concurrent updates on the same line. The data are sorted in alphabetical order of the row-keys. A row range is called tablet which is split dynamically. Tablets are the units for load balancing – very important for the horizontal scalability. In this way reading of small tablets becomes more efficient and only the communication with a small amount of machines is usually needed.

Column keys are grouped into sets which are called column families. They are the basic unit of the access control and also of the memory accounting. Values which are stored in one column family are usually of the same type. Every column key can use the column families. There is usually a small number of column keys, which are rarely changed.

Every column can contain different versions of the same data. They are indexed with timestamps. BigTable can assign timestamps automatically, but the client can also assign the timestamps by itself. Different versions are stored, so that always the newest version is read at first.

Google offers the BigTable API to access the database. The API provides functions to create and to delete tables and column families. In addition to that, it offers functions to change clusters, tables, column family metadata and access control rights. Clients can write or delete values in the tables. Furthermore they can search for certain values and iterate over a certain set of values. There are different functions to manipulate the data. Single-row transactions are used for atomic read-modify-write sequences on stored data with a certain single row key. Furthermore BigTable allows using columns as integer counters. Finally BigTable provide the execution of scripts which are offered to the client.

BigTable can be used in MapReduce (which will be explained later). The API offers different wrappers, which provide the usage of BigTable as an input/output source.

BigTable is based on various components of the Goggle infrastructure. It uses GFS to store log and data files. Furthermore it needs a cluster management system to control jobs, to manage resources on divided machines, to handle machine failures and to monitor the machine status.

BigTable consists of three main components: a library, which every client has to contain, a master server and several tablet server which can be added or deleted dynamically.

Document stores: Document stores can store any text in the form of documents. This allows for a search based on the document content. An example for such a document is shown in Fig. 2. A query like “Vorname” = “Wallace” would provide only documents, which contains the attribute “Vorname” with the value “Wallace”.

```
"Vorname": "Wallace"
"Adresse": "62 West Wallaby Street"
"Interessen": ["Käse","Cracker","Mond"]
```

Fig. 2 Example of a JSON document

The main advantage of document stores over traditional databases is a less strong structure. Attributes can be added or removed more flexibly [7], [9].

CouchDB (http://couchdb.apache.org/) is an example of document store, where documents with any syntax can be stored. Documents are JSON documents here. Such data structures are equivalent to tuples of relational databases. Every document gets a document-ID and a revision-ID for indexing. Then they are stored in B-trees. For each update the revision-ID will adjust. In this way an incremental search of the changes is possible.

CouchDB is oriented towards BigTable’s database engine and thus at the access control over the MapReduce algorithm. CouchDB relies on proven principles. The developers are focusing on the easy use of the database. It takes under consideration that a network connection is not open all the time and that there can be errors in distributed systems.

CouchDB supports all ACID properties. But reading access is implemented with multi-version concurrency control (MVCC), which ensures the replication of changes on other nodes. Every user gets a consistent snapshot of the database from the beginning till the end of the read operation. Thereby MVCC controls the access to the data [2].

As earlier mentioned, the documents are JSON documents. JSON-Objects consist of a comma-separated list of properties.

Each property is a key-value pair in which the value can also be a property. Basic types include, e.g., objects, arrays, strings etc. CouchDB also offers adding attachments to the documents.

If the data between applications are exchanged in JSON documents, this can also be used for storage and so increasing the performance for the involved applications.

The integrated view model enables the aggregation and representation of the documents. Views can be created dynamically and they have no impact on underlying data.

CouchDB offers the possibility to replicate data incrementally on multiple nodes with bidirectional conflict detection and conflict management. In this way reading data can be parallelized. The continuous replication is either triggered by an application or by the database system itself. The distribution of CouchDB is done by replication.

Replication is also the foundation for scaling. CouchDB does not provide partitioning and sharding. Those features need an additional binding to additional open-source frameworks like CouchDB-Lounge.

Key-value stores: The principle of key-value stores is simple. A key has a value, for example an arbitrary string.

These databases can be divided into two subgroups: in-memory-variant and on-disk-version [7]. The first option ensures a high performance. It maintains the data in memory. Through that the database can be used as a distributed cache memory system. The second option stores the data directly on disk. Through that the database can be used as a data storage.

Redis (http://redis.io/) is an example of a key-value store of the subgroup on-disk-variant. It is fast because all data are stored in RAM. It synchronizes with disk from time to time. This also means that a lot of RAM must be available.

There is a similarity to column-oriented databases because Redis also store lists, sets and hashes, in addition to strings. Lists are important because the functions LPUSH and RPUSH allow writing to the database with constant complexity (so very quickly). Sets are important because they allow for many set operations and thus rich queries.

Redis has two modes of execution: snapshotting and append only file [2]. In the first mode, by default the configuration of Redis provides that all data are stored onto RAM and in certain time intervals onto disk. In case of a crash, the past operations are reloaded to restore the original state. The database admin can configure the storage interval (maximum number of writings, time limit). The second mode writes all data to disk. During a restart all operations can be executed again to restore the previous state (pre-cash state).

Redis also offers a compression (or compact) mode, which restores the last state of the database in a separate process. After that, it replaces the actual file through the new file.

Redis offers a rather abstract API. But many Client-APIs for different languages like Ruby, Python, PHP, etc. are available.

For replication, Redis works with master-slave architecture. There can be one master and any number of slaves. Slaves can be connected in row or in series. This gives the possibility of different useful architectures. For example, the configuration of the master can specify that there is no writing to disk, but the slaves store the data. In this way the slaves can respond to very complex queries and relieve the master simultaneously.

C. Map Reduce Algorithm

Hadoop (http://hadoop.apache.org/) is a free, Java-based framework for large-scale systems. One fundamental part of this framework is an implementation of the MapReduce algorithm. Hadoop has been developed to work effective with large clusters (up to 10,000 nodes). One of the biggest Hadoop clusters worldwide is used by Yahoo. It consists of approximately 4,000 nodes with 32,000 cores and 16 petabytes of data. Analyzing and sorting of a data block with 1 petabyte file size takes about 16 hours in this cluster.

Hadoop basically consist of the two main components: Hadoop Distributed File System and MapReduce algorithm.

Hadoop Distributed File System: Hadoop Distributed File System (HDFS) is a distributed scalable and highly available file system, which is necessary for processing extremely large amounts of data. The MapReduce algorithm needs such a file system in order to be robust and scalable. Therefore, Hadoop provides HDFS, which is based on the GFS implementation. The architecture of HDFS (see Fig. 3) is based on one master node (NameNode) and many slave nodes (DataNodes). The master node's main task is to manage the data notes. In addition to that, the master node can simultaneously work as an additional data node, too. Primarily the master node does not store any real data. Rather, it stores only metadata, which describe the file system itself. Therefore the capacity of the HDFS cluster is limited by the memory size of the master node. The HDFS splits files into fragments and distributes them within the cluster. By default the fragment is stored twice on the same rack and once on another rack for reliability, so that even if an entire rack fails at least one of the three fragments is available. This kind of distribution allows parallel access to the stored data and increases reliability and access speed as a positive side-effect. Optionally, the data integration can be maintained by a checksum and in cause of a potential data corruption it redirects to an alternative intact fragment.

Beside all beneficial aspects of this architecture, there is one major disadvantage, which lays in the occurrence of just a single master node. This set up creates a single point of failure for file system access. Any attempt to access the file system will not be possible in case of a master node downtime. The cause of a low failure probability, unlike e.g. to raid memory models, HDFS uses a flat memory model. This model reduces recovery time after a failure and thereby, it reduces the risk of data loss due to multiple errors.

Another task of the master node is to delegate tasks to the place, where the task-associated data is stored. This approach reduces the network load dramatically, since data must not first be transferred to its processing node. It can be edited directly, ideally on the same machine or in the same rack.

Since each access to the file system results in a first-

instance master node request, since the master node is responsible for managing file metadata. These metadata do contain information about file fragmentation and file system addressing. After that, the master node delegates the request of the clients to the nearest data nodes, which also starts the file transfer by itself. This division ensures that the master node is not overloaded and quickly available for other tasks. At the same time this division allows a distributed data traffic between the data nodes and the requesting client.

For file transfer from a client to HDFS there is a similar sequence. The client tells the master node that it wants to put some files to HDFS. The master node creates some entries in its metadata storage for the new files and allows the client to start with the transaction. Because of the internal fragment replication inside the HDFS cluster, the files will be fragmented into byte blocks on the client side before the real transaction into HDFS starts. Once a block of bytes has reached a certain size, the client tells the master node that it is ready for some transaction. The master node answers with a specific location for the byte block inside HDFS. At the same time the master node sets a list of data nodes, which should receive the replicas of the byte block. The real transfer is often referred as a kind of pipelining. The client transmits the block of bytes to the data node, which it gets from the master node right before. If the byte block has completely arrived at the target data node, the node by itself starts to replicate this block to the next node on the list generated by the master node. This process is repeated several times (how often depends on the configuration of HDFS) and remains invisible for the client, so that the client can already transfer the next block of bytes.

In large-scale systems, hardware failures are rather seen as a pre-calculated state than as a failure. Because of that the data nodes send heartbeats to the master node at configurable intervals, which can determine whether a data node is still available. In case of a permanent data node downtime, the master node instructs the data nodes, containing the failure nodes duplicated data, to replicate the data within the cluster to ensure a high data availability again.

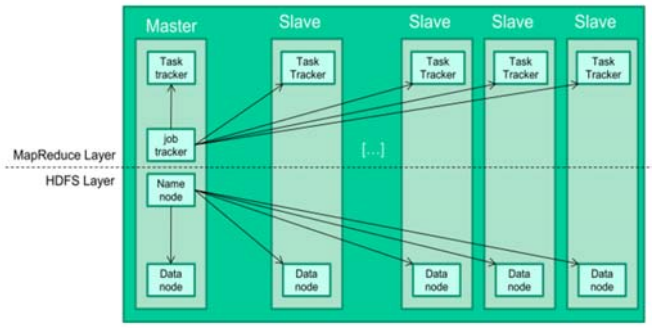

Fig. 3 Architecture of HDFS [12]

MapReduce Algorithm: Hadoop's implementation of the MapReduce algorithm, which uses HDFS, is very similar to Google's MapReduce implementation, which uses GFS. It is intended for massive parallel computation with large amounts of input data. The implementation of the MapReduce algorithm is ideally based on the architecture of HDFS (see Fig. 3) so that it has also one master node (the job tracker) and many slave nodes (the task trackers). The job tracker receives the work tasks and delegates them to the task trackers.

Basically the MapReduce algorithm consists of the two components Map and Reduce. The MapReduce algorithm divides the calculation of a certain task into two phases. The first phase (Map) calculates intermediate results of the task. After completion of the Map phase, the Reduce phase starts. The calculation in this phase is based on results of the first phase. Both phases are executed massively parallel, which makes the MapReduce algorithm high efficient, especially in relation to large amounts of data.

A simple example to illustrate the Map and Reduce phases is the determination of word frequencies within a text file. In this example, the Map phase would generate for each word in the text file a list with a structure like `(word [1,…, 1])`. If no list for a specified word is found, a new list will be generated for the given word. Otherwise, if there is already a list, another "1" is appended at the end of the list. Each "1" represents a hit for the given word in the text. After the Map phase has finished, the intermediate result storage is filled with n lists like `(word [1,…, 1])` for n different words in this text. The subsequently introduced the Reduce phase uses the previously filled intermediate result as input and counts for each list and counts the occurrence of "1", which represents the frequency of this word for each list. After all lists are processed by the reduce phase, the MapReduce algorithm has finished its task and returns with n lists of the form `(word [Frequency of the word])` for n different words [10].

As mentioned earlier the idea of HDFS was that data operations taken place as near as possible to the data needed for the operation. This idea also takes place in the architecture of the MapReduce algorithm (see Fig. 4), which is also based on the architecture of HDFS. Basically a MapReduce cluster consists of one task tracker and many job trackers. Similar to the name node at HDFS, the task tracker is used for managing task inside the MapReduce algorithm. It is often installed together with the name node on the same machine, because it has to know the location of every stored data at any time. Most of the calculations are based on large amounts of data, for this reason it becomes important to decrease delays through I/O and network processing. This is maintained by the task tracker delegates the map and reduce tasks to the worker nodes so that they are close to the stored data. At the same time, the task tracker has to look after all other worker nodes and their intermediate results and I/O data.

To reduce the complexity of a MapReduce task, it is possible to specify the input and output reader in order to fragment a complex task into several easier MapReduce jobs with works together.

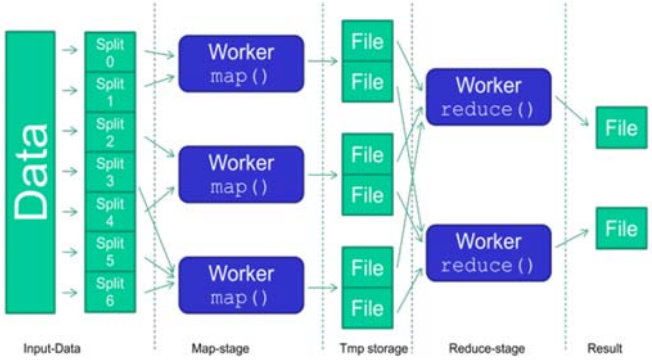

Fig. 4 Architecture of Map Reduce algorithm [12]

D.Caching

One aspect of a system's performance is to have the right data at the right place at the right time. This applies to both hardware and software. Caching mechanisms are a common and efficient way to temporarily store data in a fast memory near the location they are needed (presumably). A prominent example on the hardware side is multi-level caching that keeps data as close as possible to the CPU. In magnetic hard disk drives, caching is used to avoid unnecessary slow mechanical access. On the software side, caching is heavily used inside browsers. Resources such as pictures are stored on the local machine, which reduces network traffic and bypasses the bottleneck of a remote connection.

In large-scale systems caching can increase the overall performance. As explained above, large-scale systems consist of distributed file systems, NoSQL databases and algorithms that work on huge amounts of data. These components may all be considered as bottlenecks, even more if they have to play together to perform a given task. Therefore, the avoidance of performing a certain complex query more often than necessary is more crucial than in traditional systems.

One major problem with all caching mechanisms is how to keep the cached data up-to-date. If caches are updated permanently, there may be no gain of performance anymore, due to the produced overhead. On the other hand, an application may be hardly able to benefit from out-of-date data that are provided by infrequently updated caches. It depends on the type of query and application, whether a big-scaled caching system should be used. Considering a stock exchange system that heavily depends on up-to-date data, an intensive caching might not be applicable, due to the time-sensitiveness of the data involved in the process. In social web application like Facebook actuality of data is less critical. Tasks like updating a message board or sharing a picture do not have to be performed in real time, but may benefit from caching when data are frequently accessed in the future.

Memcached (http://memcached.org/) is a frequently used caching solution that can be easily integrated into large systems. It is an open source and has been introduced in 2003 to improve the performance of dynamic web applications [17]. Mainly used to cache frequently requested websites to avoid accessing the database, Memcached can be used to cache arbitrary data, so called chunks. Basically Memcached can be regarded as a high performance, distributed memory object model, a "short-term memory" for applications.

Internally, Memcached uses a vast hash table. It can be distributed across multiple machines while its keys can be up to 250 bytes long. Servers keep the entire hash table inside the RAM to avoid performance leaks caused by hard disk drive accesses.

Memcached is based on a client/server architecture. The server manages the cache. In detail, there may be several servers, which work independently from each other. Servers do not know other servers and there is no synchronization or broadcasting between them.

Every component in the system, which provides data to the cache, may act as a client. Every client knows all servers. When asking for a certain chunk, a client calculates the chunk's hash value to determine the server, which is caching that chunk. The server itself has to calculate its own hash value from that key to get the location of the chunk. This applies for both reading and writing. If all clients share one hashing algorithm, they are able to share one cache.

Memcached combines the memory of several servers to one big virtual memory, as shown in Fig. 5. Many small, server-depended memories are inflexible and hard to share. A shared, logical cache provides easy sharing of data among clients and allows horizontal scalability by just adding new servers or memory. Latter are available as dedicated hardware units. This allows independent scalability of calculation power and cache-size.

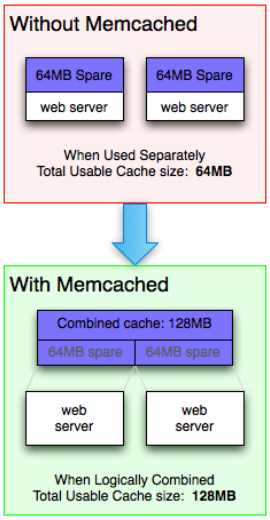

Fig. 5 Usage of Memcached [17]

Like every memory system, Memcached is limited in its size. Once the cache is completely filled, chunks have to be removed. Here, a common last-recently-used algorithm is used to ensure the cache stays up-to-date as much as possible. Therefore, Memcached must not be treated as an in-memory database, because it cannot provide permanent persistence. Anyway, a persistence enabled variant of Memcached called

MemcacheDB is available.

Since Memcached is a hash-based key-value store, it does not provide any kind of queries, nor set operations.

In its standard implementation, Memcached does not provide any security features such as encryption or authorization. This is because Memcached is optimized for performance and security features would take some of it. This may not be a disadvantage in most cases, since caching is supposed to be run in a DMZ. If needed anyway, there is an optional Simple Authentication and Security Layer (SASL) support, when recompiling Memcached [17].

Memcached is used by many large-scale systems such as Wikipedia, Flickr, Youtube, Twitter, Facebook among others [16]. It provides API for most popular languages, such as Java, C#, C++, Pearl, etc. In the Java world it is well integrated using an advanced programming interface and integration into Spring and Tomcat sessions.

Another caching solution that may be used in large-sale systems is Redis (discussed before).

IV. Examples of Large-scale Systems

Playfish and Facebook are two important examples of large-scale systems.

A. Playfish

Playfish [13] provides social games for platforms like Facebook, MySpace and iPhone. There are 10 million active users a day and hundreds of server machines. And more and more games are released so that it will continue to grow. Here are some facts about the architecture of Playfish.

1. Playfish provides social games, meaning that there is interaction between the users. But the games are *asynchronous*. The players can play at different times. In this way it is possible that the players play on their own client.
2. Each individual game is easy to scale because there are only a few users. But the whole system is harder to scale because of the *sheer number of users* actively playing games.
3. There is a rapid expansion of social games so the architecture has to deal with a *continuous stream of new users.*
4. At the beginning no one knows how successful a single game will be. If the game becomes successful there must be a possibility to *expand immediately*.
5. A smart client can decrease a server's read access, so most of the *database activity is to write heavily.*
6. The whole system has the need to be *scalable in different dimensions.* "Playfish needs to scale up to support more users, more games, more data per user, more accesses per user, more development staff" [13].

All these points have to be addressed in the architecture. For that, Playfish uses the following scaling strategies:

1. Playfish was cloud-based from the very beginning. They launched their first game as a beta on Amazons Elastic Compute Cloud EC2. The cloud can solve a high rate of issues, which are related to a changing demand of resources. Depending on the success of a game they can spin up more resources or they can simply give the instances back. The IaaS (Infrastructure as a Service) is profitable for Playfish.
2. On the server side they use SOA (Service-oriented Architecture) as an organizing structure of the system. Each game is a separate service and each service can be released independently.
3. Playfish went to a sharded architecture. This is the only real way to scale up write activity. For data storage they use the key/value approach with a MySQL Database. The values are stored in a BLOB format and the data are sharded across different MySQL clusters. Each cluster has its own master and read replica. Why do they not use a NoSQL database instead of MySQL? The developers thought about this option. Till that point of time they used MySQL. So they have the requirements for which NoSQL databases are made, but they have also a running solution. For scaling they needed something like sharding but then many SQL features like indexing will not be available. Furthermore they would give up flexibility of access patterns when they use NoSQL. So instead of a NoSQL database they continued to use MySQL. With the BLOB format they can misuse MySQL as a key-/value-store.
4. Asynchronicity is also an important point for scaling at Playfish. A server's write access is handled asynchronous. On this way they try to hide the latency from the user as much as possible.
5. Furthermore they use the proposed Map Reduce Algorithm supported by Amazon.

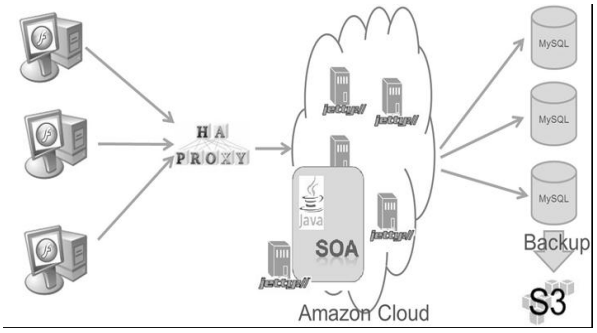

Fig. 6 Architecture of Playfish

Fig. 6 shows the architecture of Playfish. On the client side they use Flash. With the number of users the number of client side CPUs and resources grow. That's why they put as much processing as possible to the client. The writing of changes to the server is asynchronous. The communication between the Flash client and the Java server is ensured by a service level API.

The backend is implemented in Java. For the architecture they use SOA. Therefore Playfish uses Jetty Server as Java Application Servers. They are stateless, which simplifies the

deployments and upgrades, and improves the availability. As previously described, they use MySQL as their data tier. They store the data in BLOBs instead of normalization into rows because they want to optimize the size of user records to fit more users in memory. Because of their key/value-strategy there is one database record per user. 60% of the workload is writing. They use sharding to get more performance.

Because of the Flash clients, Playfish does not need scaling techniques like Memcached. Most of what would be cached by Memcached is cached in the client. The backups from the databases are performed on S3.

The complexity of the architecture is an advantage. The developers are trying to transfer the system's workload to the clients so that they do not have to deal with any trouble to upscale on the server side on this point. Every game is a service so each game can be integrated into the architecture easily. They use the Amazon cloud for the Jetty server so they can scale out very easy when they have a rising demand on system resources. On their database tier they use MySQL databases. Within time they gained expert experience in the field of MySQL which makes it to their designated database engine. To shard the data tier they use MySQL as a key-/value-store. So they found a good way to build a large-scale system with well-known widely used techniques.

B. Facebook

With over 500 Million users, Facebook (http://facebook.com) is the largest social network in the world. Each month the page is visited more than 200 billion times. This includes also over 15,000 websites which uses Facebook Connect e.g. the Like-Button. Due to this high access rate it is not very surprising that Facebook causes about 10% of the worldwide internet traffic today. Facebook does not publish much information about its IT architecture but some key data are transparent to the public. Facebook scales between its own nine data centers. Among these data centers more than 60,000 servers are distributed. Based on this fact, Facebook can certainly be viewed as a large-scale system.

The architecture of Facebook itself is fundamentally based on a LAMP architecture (see Fig. 7). For this reason, Facebook's operational IT infrastructure is based on open source software. It is founding on Linux, Apache, MySQL and PHP (LAMP) containing one of the biggest MySQL clusters in the world [20].

Facebook does not use the basic LAMP components out of the box, mostly it uses high specialized versions, which are only built by and for Facebook. Since a compiled code usually runs faster as runtime interpreted code like PHP, Facebook often codes critical functions in C++ code and uses them as a RPC service. RPC services often use previously described technologies like databases and frameworks such as Hadoop, Cassandra, Hive or Scribe [18].

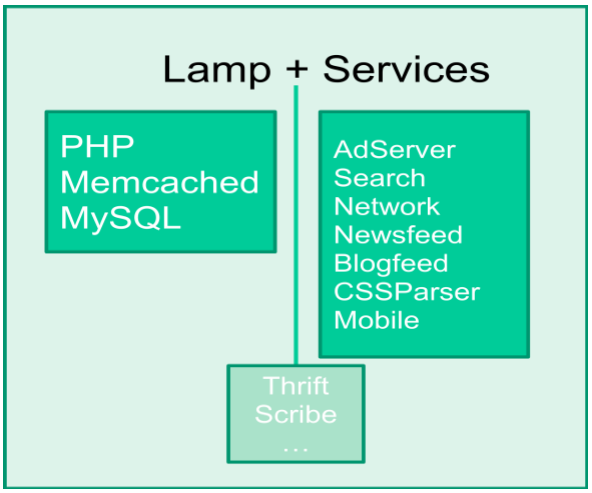

Fig. 7 Basic architecture of Facebook [20]

PHP: Facebook uses PHP as a preferred basic programming language for their basic framework, because PHP lives from its active developer community. Other advantages are the good library support for web applications. Also the dynamic typing and interpretation of PHP as a script language is an advantage, too [20].

Memcached: This is a high-performance, distributed memory object caching system, which reduces the load on the database. This benefit is caused by caching the result of frequent queries. However, Facebook has thousands of servers and every server runs hundreds of Apache processes, which all communicate with Memcached. Memcached usually sets a buffer for each TCP connection, which allocates memory. Since Facebook's IT infrastructure has to process a high rate of queries in a really short time, Memcached has a high demand on system memory just for connection buffering. To deallocate this memory, Facebook invented a "per-thread" shared buffer pool for UDP and TCP sockets. These and other specializations from the Facebook developers are allowing Facebook's Memcached to answer up to 200,000 UPD queries per second with an average latency of 173 milliseconds. Comparing Facebook's Memcached to its standard version, which can handle for about 50,000 queries per second, it is obvious that Facebook's customization of Memcached came along with a big performance improvement [11].

MySQL: The main fact why Facebook uses MySQL is the good data integrity. MySQL is using checksums and has other possibilities to ensure data integrity. However, Facebook does not take advantage that MySQL is a relational database. Facebook's MySQL does not have any joins in its code. Every value has its own UID. Because of scaling is easier at the web tier, Facebook does not have much functionality in their database servers. The databases scale themselves across multiple data centers and they are using Memcached proxies to take care of deleting or updating all copies of the data. To take benefit of this advantage, Facebook has extended MySQL language to include instructions for the Memcached proxies [20].

Due to the fact that Facebook will not be so successful without a vibrant open source community, Facebook often publishes its own experiences and projects as open source in order to help other developers.

V. Conclusion

Large-scale systems are systems that exceed traditional software in various dimensions. Large-scale systems are systems that cannot be build, because they exceed the way software is engineered today, meaning that they are systems of systems [14]. The main challenge when building a large-scale system is to provide an architecture that allows the whole system or its parts to grow into dimensions that are not known at the time of birth of the system. With today's engineering technology this is achieved by using distributed systems that are able to scale horizontally on every level of the architecture.

This article showed how horizontal scaling approaches are used in file systems, databases and algorithms to let large-scale systems grow dynamically. The investigation of the Google File System and BigTable, Redis and CouchDB database implementations proved that even the very basic elements of a a large-scale system are highly optimized for certain tasks. Parallelized algorithms such as MapReduce require a fast underlying hardware and software structure to be able to show their performance. Caching is one mechanism that is used to get around decelerating network connections.

Large-scale systems are always complex and adopt solutions, which are assembled from optimized components and systems. Here the choice of technologies depends on the domain of the systems.

The inspection of today's biggest and most frequented applications such as Facebook and Playfish has shown that large-scale systems are already a reality today. Here again hardware and software are stressed to their limits, which cannot be solved by using today's methods and technologies.

In the future, further research will be required to meet the demands of tomorrow's large-scale systems. This includes the areas of parallel-working hardware, optimized algorithms, engineering technologies as well as the development process and how we think about putting systems together [15].

Acknowledgment

Irina Astrova's work was supported by the Estonian Centre of Excellence in Computer Science (EXCS) funded mainly by the European Regional Development Fund (ERDF). Irina Astrova's work was also supported by the Estonian Ministry of Education and Research target-financed research theme no. 0140007s12.

We would like to thank Mats Lennart Henke from the University of Applied Sciences and Arts Hannover, Germany, for his help in preparing this paper.

References

[1] D. Masak, "SOA?: Serviceorientierung in Business und Software" Springer, 2007.
[2] S. Edlich, A. Friedland, J. Hampe, B. Brauer, „NoSQL - Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken 2010", Hanser, 2007.
[3] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fike, R. Gruber, "Bigtable: A Distributed Storage System for Structured Data", Goole Inc., 2006.
[4] http://www.oser.org/~hp/bsyII/node6.html, "Skalierbarkeit", 2007.
[5] http://blog.namics.com/2009/05/skalierbare-dat.htm, „Skalierbare Datenbanksysteme: ACID versus BASE", 2009.
[6] http://nosql-database.org/, "NoSQL - Your Ultimate Guide to the Non - Relational Universe!", 2011.
[7] http://www.heise.de/open/artikel/NoSQL-im-Ueberblick-1012483.html, „NoSQL im Überblick", 2010.
[8] http://www.gi.de/service/informatiklexikon/informatiklexikon-detailansicht/meldung/spaltenorientierte-datenbanken-267.html, „Spaltenorientierte Datenbanken", 2010.
[9] http://eliteinformatiker.de/2011/05/18/nosql-document-store-couchdb-mongodb/, „NoSQL: Dokumentenorientierte Datenbanken (CouchDB, MongoDB)", 2011.
[10] Cloud Computing: Web-basierte dynamische IT-Services (Informatik im Fokus) , 2. Auflage, Springer ,ISBN 978-3-642-18435-2.
[11] "Scaling memcached at Facebook", Paul Saab, December 2008, Facebook Engineering, http://www.facebook.com/note.php?note_id=39391378919.
[12] Oliver Fischer, "MapReduce: Programmiermodell und Framework", April 2010, http://www.heise.de/developer/artikel/Programmiermodell-und-Framework-964823.html.
[13] "Playfish's Social Gaming Architecture - 50 Million Monthly Users And Growing", http://highscalability.com/blog/2010/9/21/playfishs-social-gaming-architecture-50-million-monthly-user.html, 2010.
[14] Richard P. Gabriel, "Can't Be Built" IBM Research, 2007.
[15] Eeles2008, Peter Eeles, Presentaion on "Architecting Large-Scale Systems, IBM, 2008.
[16] Paul Saab, "Scaling memcached at Facebook", 2008, via Facebook.com
[17] Brad Fitzpatrick , "Distributed Caching with Memcached", 2004, on http://www.linuxjournal.com/article/7451?page=0,1.
[18] Realtime Hadoop usage at Facebook: The Complete Story, Dhruba Borthakur, http://hadoopblog.blogspot.com/search/label/hadoop%20and%20facebook.
[19] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung," The Google File System", Google Coorp., 2003.
[20] Facebook: Science and the social Graph, Aditya Agarwal, May 2009, http://www.infoq.com/presentations/Facebook-Software-Stack.
[21] AmazonStatistics; Jeff Barr, "Amazon S3 Blog" July 2011 http://aws.typepad.com/aws/2011/07/amazon-s3-more-than-449-billion-objects.html.